

AD-A245 807

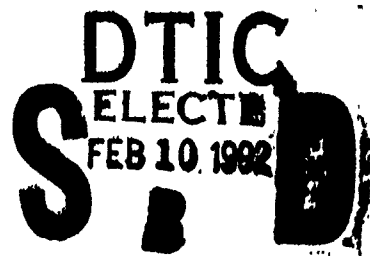


# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



# THESIS



**DESIGN AND IMPLEMENTATION  
OF A  
CONCRETE INTERFACE GENERATION SYSTEM**

by

Randy James Rachal

December 1990

Thesis Advisor:

Valdis Berzins

Approved for public release; distribution is unlimited.

92-03047



02 2 00 07A

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

1. REPORT SECURITY CLASSIFICATION <b>Unclassified</b>			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT <b>Approved for public release; distribution is unlimited</b>		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION <b>Naval Postgraduate School</b>	6b. OFFICE SYMBOL (If applicable) <b>37</b>	7a. NAME OF MONITORING ORGANIZATION <b>Office of Naval Research</b>			
6c. ADDRESS (City, State, and ZIP Code) <b>Monterey, CA 93943-5000</b>		7b. ADDRESS (City, State, and ZIP Code) <b>800 N. Quincy Street Arlington, VA 22217-5000</b>			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) <b>The Design and Implementation of a Concrete Interface Generation System</b>					
12. PERSONAL AUTHOR(S) <b>Randy J. Rachal</b>					
13a. TYPE OF REPORT <b>Master's Thesis</b>	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) <b>December 1990</b>		15. PAGE COUNT <b>94</b>	
16. SUPPLEMENTARY NOTATION <b>The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.</b>					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	<b>Automatic Code Generation, Formal Specifications, Attribute Grammars</b>		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) <b>The purpose of this thesis is to design and implement a concrete interface generation system. The concrete interface generator is a software system which takes a formal specification as input and generates the specification part of an Ada implementation as output. Attribute grammars and fourth-generation language tools have been used in the implementation of this system. Spec, a formal language for writing black-box specifications for large software systems, was used as the input for the concrete interface generation system. Ada was chosen to be the computer language generated by the system. This thesis implements a subset of the Spec language, discusses the design methodology used in its implementation, and presents guidelines for the mapping of Spec to Ada. Included is a listing of the Spec grammar, the concrete interface generator systems source listing, a sample of input used to test the system, and resulting output.</b>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION <b>Unclassified</b>		
22a. NAME OF RESPONSIBLE INDIVIDUAL <b>Dr. V. Berzins</b>			22b. TELEPHONE (Include Area Code) <b>(408) 646-2461</b>	22c. OFFICE SYMBOL <b>CS/Be (52Be)</b>	

Approved for public release; distribution is unlimited.

**Design and Implementation of a  
Concrete Interface Generation System**

by

Randy James Rachal  
Lieutenant, United States Navy  
B.S., Northwestern State University, 1982

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**


from the

**NAVAL POSTGRADUATE SCHOOL  
December 1990**

Author:

  
Randy James Rachal

Approved by:

  
Valdis Berzins, Thesis Advisor

  
Leigh W. Bradbury, Second Reader

  
Robert B. McGhee, Chairman,  
Department of Computer Science

## ABSTRACT

The purpose of this thesis is to design and implement a concrete interface generation system. The concrete interface generator is a software system which takes a formal specification as input and generates the specification part of an Ada implementation as output. Attribute grammars and fourth-generation language tools have been used in the implementation of this system. Spec, a formal language for writing black-box specifications for large software systems, was used as the input for the concrete interface generation system. Ada was chosen to be the computer language generated by the system. This thesis implements a subset of the Spec language, discusses the design methodology used in its implementation, and presents guidelines for the mapping of Spec to Ada. Included is a listing of the Spec grammar, the concrete interface generator systems source listing, a sample of input used to test the system, and resulting output.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## TABLE OF CONTENTS

<b>I. INTRODUCTION.....</b>	<b>1</b>
A. OBJECTIVES .....	2
B. RESEARCH QUESTIONS.....	3
C. ORGANIZATION OF STUDY .....	4
<b>II. BACKGROUND.....</b>	<b>5</b>
A. SPEC—A FORMAL SPECIFICATION LANGUAGE.....	5
1. Event Model.....	5
a. Modules .....	6
b. Messages.....	8
c. Events.....	9
d. Alarms.....	9
2. Some Other Components of Spec .....	9
a. Definitions and Instance Modules .....	9
b. Concepts .....	10
c. Semantic Issues.....	10
B. ATTRIBUTE GRAMMARS.....	10
C. KODIYAK APPLICATION GENERATOR.....	11
1. Format.....	13
2. Comments.....	13
3. Lexical Scanner Section .....	13

4. Attribute Declaration Section.....	15
5. Attribute Grammar Section.....	16
6. Using Kodiyak.....	17
D. TOOLS IMPLEMENTED WITH ATTRIBUTE GRAMMAR	
TOOLS.....	18
1. Specification Language Type Checker.....	18
2. Language Translator For The Computer-Aided Prototyping System (CAPS).....	18
3. Test Result Classifier.....	18
4. Specification Pretty Printer.....	18
<b>III. DESCRIPTION OF THE DESIGN AND IMPLEMENTATION....</b>	<b>20</b>
A. THE PROBLEM.....	20
1. Why Ada Was Chosen .....	20
2. Packages.....	21
3. Limitations of this Implementation.....	21
4. Intended Usage.....	24
B. THE DESIGN.....	24
1. Mapping Spec to Ada.....	24
2. Translation Templates.....	25
C. IMPLEMENTATION.....	29
1. Attribute Characteristics.....	30
2. Attribute Dependency Diagram .....	31

3.	Attribute Definitions.....	31
a.	ada_specification .....	32
b.	generic_parameters.....	32
c.	name.....	32
d.	subprogram_declarations .....	33
e.	exception_declarations .....	33
f.	input_parameters .....	33
g.	return_type .....	33
h.	output_parameters.....	33
i.	num_output_variables.....	34
j.	parameter_name.....	34
k.	parameter_type .....	34
l.	%text.....	34
D.	SYSTEM GENERATION PROCEDURE.....	35
E.	SAMPLE INPUT AND OUTPUT.....	35
1.	Non-Generic Example .....	35
2.	Generic Example.....	36
3.	Multiple— Messages, Inputs .....	37
4.	Multiple Outputs, Exception Declarations .....	39
IV.	CONCLUSIONS .....	40
A.	EVALUATION OF THE DESIGN.....	40
B.	EVALUATION OF THE IMPLEMENTATION .....	41
C.	FUTURE WORK.....	41

APPENDIX A	GRAMMAR FOR THE FORMAL LANGUAGE SPEC.....	42
APPENDIX B	KODIYAK SOURCE CODE TO CREATE THE.....	53
APPENDIX C	ADACI USER'S GUIDE .....	75
LIST OF REFERENCES .....		79
INITIAL DISTRIBUTION LIST.....		81



## **LIST OF FIGURES**

1.1	System Creation .....	2
1.2	System Use .....	3
2.1	Example of a Spec Function.....	7
2.2	Example of a Spec Machine.....	7
2.3	Spec Example of a Type.....	8
2.4	Grammar Symbol and Attribute Relationsh.....	12
2.5	Sample Lexical Definitions.....	14
2.6	Sample Attribute Declarations.....	16
2.7	Syntax of Kodyak Rules .....	17
3.1	Complete Ada Package.....	22
3.2	Spec Description of square_root Function .....	23
3.3	Procedure Using square_root_pkg.....	23
3.4	Package Template .....	26
3.5	Function Declaration Template.....	27
3.6	Machine Declaration Template .....	28
3.7	Procedure Declaration Template.....	28
3.8	Procedure Parameters Template.....	28
3.9	Type Template .....	28
3.10	Production Rule With Attributes.....	29
3.11	Attribute Dependency Diagram .....	30
3.12	Non-Generic Example .....	36

3.13	Generic Example.....	37
3.14	Example of Multiple Messages and Multiple Inputs.....	38
3.15	Example of Multiple Outputs and Multiple Exception Declarations .....	39
C.1	Generic Example, Input: Spec.....	77
C.2	Generic Example, Output: Ada Specification .....	78

## **I. INTRODUCTION**

Due to the ever-increasing capability of computer hardware coupled with the ever-decreasing cost of the computer hardware, there is a great pressure being placed on the development of today's software systems. The software systems being asked for in today's environment are becoming larger and much more complex. Some systems being developed have more than a million lines of code. A software development organization numbering 100 individuals may take years to complete a large software project.

Of course, the development cost of such large projects is going to be enormous. Since more processes are being automated and projects are taking longer, the pool of available personnel has dwindled, thus driving wages up.

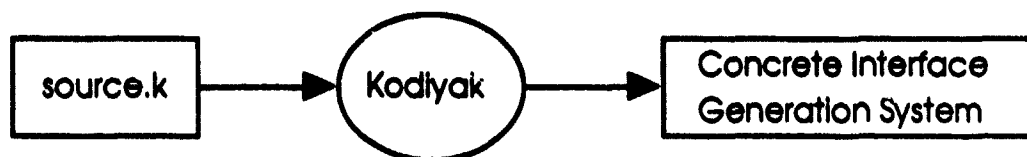
As a result of these problems, a rapidly growing area of software engineering is the development of tools to aid in the software development process. One tool developed by Berzins [Ref. 1] is the formal specification language Spec. Spec is used for writing black-box specifications for components of software systems in the formal specification stage of software development. The black-box specifications contain the major decisions in the architectural design and are produced by highly skilled software designers.

Once the design is complete, average programmers implement these Spec specifications. Due to the normal human factors involved, the implementation process is error prone and time consuming.

To help solve these problems, a software tool is needed that will reduce errors and decrease the time required to implement formal specifications. This thesis centers on the design and implementation of a concrete interface generation system that would automate part of the implementation process.

### **A. OBJECTIVES**

This thesis extends the application of attribute grammars and fourth-generation language tools to include production of a concrete interface generation system. The system is created by developing a source program for the fourth-generation language tool Kodiyak. The source program is compiled by Kodiyak, which generates the desired system (see Figure 1.1). The design methodology and guidelines for producing concrete interfaces are discussed in Section B of Chapter III.



**Figure 1.1. System Creation**

The concrete interface generator contemplated for this research is a software system which takes a formal specification as input and generates the Ada specification as output (see Figure 1.2). Once the concrete

interface generation system has been created, it does not interact with Kodiyak.

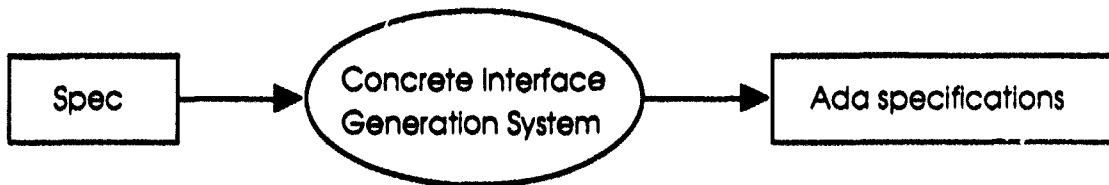


Figure 1.2. **System Use**

The concrete interface generation system will offer software engineers a cost-effective, timely, and efficient means to generate Ada specifications. Many man-hours will be saved as automatic generation replaces manual methods. The concrete interface generator will also greatly reduce the number of errors inherent in the human process of coding the Ada specifications.

## **B. RESEARCH QUESTIONS**

There are two principal research questions for this thesis. First, what are the basic concepts of and the theoretical foundations for design and implementation of a concrete interface generator using attribute grammars and fourth-generation language tools?

Second, what are the issues associated with designing and implementing a concrete interface using an attribute grammar and fourth-generation language tools? Is it a feasible means of developing a software tool such as a concrete interface generator?

### **C. ORGANIZATION OF STUDY**

The tools used to create the concrete interface generation system are described in Chapter II. Included are discussions of Spec, attribute grammars, the Kodiyak Application Generator, and several software tools previously developed.

The design and implementation are presented in Chapter III. Design methodologies, including the use of translation templates and attribute dependency diagrams, are discussed.

Conclusions are presented in Chapter IV, which also discusses suggestions for improvement of the concrete interface generation system, how the design was influenced by Kodiyak's limitations, and what extensions to the application generator would be useful when developing software tools.

## **II. BACKGROUND**

The following discussion describes the Spec grammar used in this research, attribute grammars in general, the Kodiyak application generator, and two previous tools developed using these components. The Kodiyak application generator is the software tool actually used to create the concrete interface generator. In order for Kodiyak to perform its function, a grammar of the language to be parsed (Spec) must be annotated with attributes that generate the required structures of the desired output language (Ada).

### **A. SPEC—A FORMAL SPECIFICATION LANGUAGE**

Spec is a formal language developed to make the process of describing components of large software systems more efficient and the components themselves less ambiguous for implementation. Black-box specifications are developed to describe the interfaces for each module of the software system being developed. Discussion of the event model and the Spec language extracted from Reference 2 follows. Appendix A contains a listing of the grammar for the Spec language used in this research.

#### **1. Event Model**

The event model is the semantic basis for Spec. The event model uses four primitives: modules, messages, events, and alarms. A module is a black box that interacts with other modules by sending and receiving messages. A message is a data packet sent from one module to another.

An event occurs instantaneously when a message is received by a module at a particular time. An alarm defines a particular time in a module at which temporal events will be triggered.

#### **a. Modules**

Modules are used to model software components. They are the basic building blocks of Spec. You specify a module's behavior by describing its interface, which consists of the set of stimuli (events) it recognizes and their associated responses (sets of events). All interactions must involve explicit message transmissions. There are five kinds of modules in Spec. The three most often used are:

(1) **Functions.** A function has no memory (is immutable), so its behavior is independent of the past. Completely specified function modules calculate single-valued mathematical functions, while incompletely specified function modules can behave nondeterministically. An example Spec function is shown in Figure 2.1 [Ref. 1].

(2) **Machine.** A machine does have internal memory (is mutable) so its behavior does depend on its past interactions. The behavior of a machine module is described in terms of a conceptual model of its state using a finite set of state variables. State changes can only occur at events. See Figure 2.2 [Ref. 1].

(3) **Types.** A type module defines an abstract data type. The definition must contain a value set and a set of operations on the value set. Types can be mutable or immutable. An immutable type has a fixed value set, and the operations of an immutable type cannot change



```

FUNCTION square_root{precision:float SUCH THAT precision > 0.0}
  MESSAGE(x: float)
    WHEN x >= 0.0
      REPLY(y: float)
        WHERE y > 0.0, approximates(y * y, x)
    OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: float)
    VALUE(b: boolean)
      WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END

```

**Figure 2.1. Example of a Spec Function**

```

MACHINE sender
  STATE(data:sequence(block))
  INVARIANT true
  INITIALLY data=[ ]

  MESSAGE send(file: sequence(block))
    WHEN length(file) > 0
      SEND first(b: block) TO receiver WHERE b = file[1]
      TRANSITION data = file
    OTHERWISE REPLY EXCEPTION empty_file

  MESSAGE echo(b: block)
    WHEN b = data[1] & length(data) > 1
      SEND next(b1: block) TO receiver WHERE b1 = data[1]
      TRANSITION *data = b || data
    WHEN b = data[1] & length(data) = 1
      SEND done TO receiver
      SEND done TO sender
      TRANSITION data = [ ]
    OTHERWISE SEND retransmit(b2: block) TO receiver WHERE b2 = data[1]

  MESSAGE done

  TRANSACTION transfer = send; DO echo OD; done
END

```

**Figure 2.2. Example of a Spec Machine**

the properties of the individual type instances (see Figure 2.3 [Ref. 1]). A mutable type can create and destroy type instances with internal states and can provide operations for changing them.

```

TYPE char
  INHERIT equality(char)
  INHERIT total_order(char)

  MODEL(code: nat) -- ASCII codes
  INVARIANT ALL(c: char :: ()) <= c.code <= 127)

  MESSAGE create(n: nat) -- literal 'a' = create(97) and so on
    WHEN 0 <= n <= 127 REPLY(c: char) WHERE c.code = n
    OTHERWISE REPLY EXCEPTION illegal_code

  MESSAGE ordinal(c: char) REPLY(n: nat)
    WHERE n = c.code

  MESSAGE equal(c1 c2: char) REPLY(b: boolean)
    WHERE b <=> (c1.code = c2.code)

  CONCEPT letter(c: char) VALUE(b: boolean)
    WHERE b <=> (c IN ['a' .. 'z'] | c IN ['A' .. 'Z'])

  CONCEPT digit(c: char) VALUE(b: boolean)
    WHERE b <=> c IN ['0' .. '9']
END

```

**Figure 2.3. Spec Example of a Type**

### ***b. Messages***

Messages are used to model abstract interactions. These interactions can be realized as procedure calls, returns from a procedure, Ada rendezvous, etc. Each message has four attributes: the origin (who sent the message), the name (used to identify the service requested by a normal message or the exception condition determined by an exception

message), a sequence of data values (either inputs or results), and condition (either normal or exception). The origin of a message is the event or alarm that caused the message to be sent. The reply is sent to the module that sent the stimulus, which can be determined from the message's implicit origin attribute (a feature of the event model).

### **c. Events**

The behavior of a system consists of a set of events. Each event is uniquely identified by three associated properties: a module, a message, and a time. "Time" corresponds to the time at which the module received a message.

An event can be classified as reactive or temporal. A reactive event is an event which occurs in response to an external stimulus, such as the user initiating a response from the system. A temporal event occurs when a regularly scheduled or planned event (e.g., an alarm) initiates an internal stimulus which requires a response from the system.

### **d. Alarms**

Each alarm consists of a module, a message, and a time, but an alarm causes the module to send a message to itself at the given time. Each module has a clock that measures local time which is used by the event model to control events that must happen at specific times (e.g., at 3:00 a.m. every Sunday).

## **2. Some Other Components of Spec**

### **a. Definitions and Instance Modules**

Definitions are a module type used to declare concepts that are necessary to explain the system being described. They provide access

to widely shared concepts needed by various modules. Definition modules can contain only concept definitions.

Instance modules are used to make an instance or partial instantiation of generic modules and for making interface adjustments to reusable components by hiding or changing some names.

#### **b. Concepts**

The purpose of concepts is to help decompose the specification into manageable units. Concepts represent properties that describe the system's intended behavior. In the square-root example, shown in Figure 2.1, the concept "approximates" defines what you mean by "a sufficiently accurate approximation" in terms of the generic parameter *precision*.

#### **c. Semantic Issues**

There are many semantic issues (such as scoping rules, naming constraints, and type constraints) in the Spec language that will not be discussed here. More information concerning these issues may be found in References 1 and 3.

### **B. ATTRIBUTE GRAMMARS**

Attribute grammars were introduced by Knuth when he built upon the idea of defining semantics by associating synthesized attributes with each non-terminal symbol and associating corresponding semantic rules with each production [Ref. 4]. Knuth showed that inherited attributes are useful in the cases where part of the meaning of a given construction is determined by the context in which it is used.

The development of an attribute grammar begins with a context-free grammar. Each grammar symbol is then assigned a set of attributes which are divided into two subsets: inherited attributes and synthesized attributes. Picture the node of a grammar symbol in the parse tree as a record with fields for holding information, then each attribute corresponds to one of those fields as in Figure 2.4.

The greatest use of attribute grammars has been for translating one language to another. Specifically, they are used to develop compilers and compilers-compilers (application generators), and for translating grammar specifications into executable code, such as the work presented here. Attribute grammars are extremely well suited to the development of these tools because they are easily modified when extensions are added and the attribute equations are independent of each other.

### **C. KODIYAK APPLICATION GENERATOR**

The Kodyak Application Generator is a language designed for constructing translators [Ref. 5]. It is modeled after Knuth's descriptions of attributed grammars. The language includes facilities for describing a lexical scanner, an LALR (1) grammar, and attribute definition equations [Ref. 6].

Kodyak integrates the functions of the LEX [Ref. 7] analyzer generator, the YACC [Ref. 8] parser generator, and the code necessary to compile the "C" language code into executable code. Kodyak lets the designer work in terms of equations defining attribute values, rather than with fragments of "C" code, as necessary when using YACC directly. Kodyak takes care of the interface details so the user is not bothered with them.

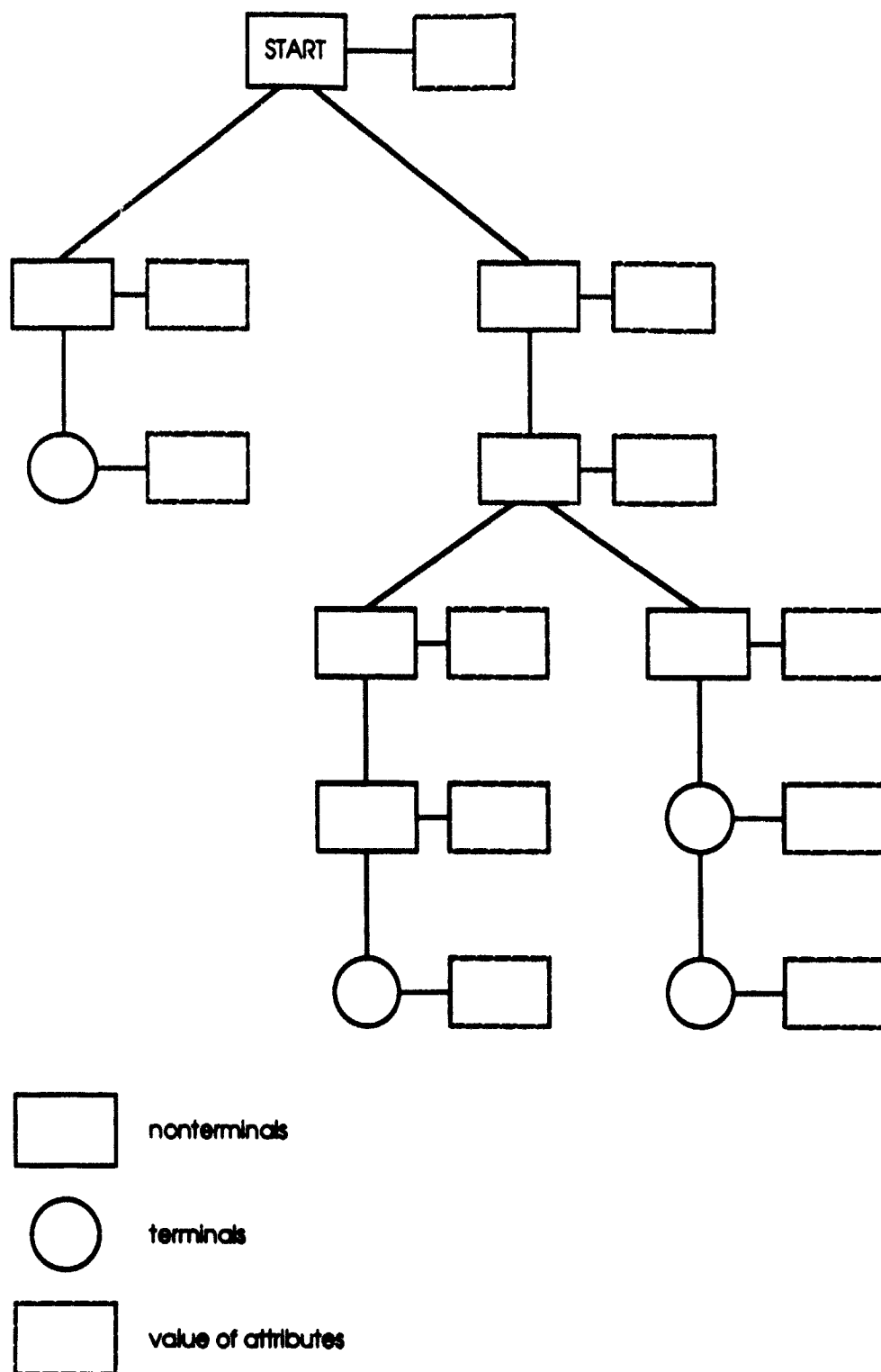


Figure 2.4. Grammar Symbol and Attribute Relationship

The discussion that follows was extracted from Ref. 6 and provides only that information required to understand the code in this research. Complete, detailed information can be obtained from References 6 and 9.

### **1. Format**

The Kodiyak program has three sections. The first section describes the features of the lexical scanner which is used to translate the source text into tokens, establish operator precedence and declare associativities for those tokens. The second section names the attributes associated with each grammar symbol and defines their types. The third section describes the grammar and the attributed equations which define the semantics of the translation. These three sections in a Kodiyak program are separated by a double percent symbol ("%%") on a line between each section.

### **2. Comments**

The exclamation point ("!") is used to start a comment which continues to the end of the line it is located on. A comment may begin anywhere on the line.

### **3. Lexical Scanner Section**

The primary function of statements in this section is to specify the terminal symbols of the grammar and how input text is to be transformed to these symbols. The secondary function is to specify a set of operator precedence to be used in conjunction with the grammar. An example of typical statements found in this section are shown in Figure 2.5. The most common form of the basic token definition is:

**TERMINAL\_NAME :REGULAR\_EXPRESSION**

The keyword "%define" introduces the definition of a class of terminal symbols. For example DIGIT, in Figure 2.5, is defined to be a single ASCII character between "0" and "9."

Terminal symbols can be described using the defined characters class symbols. For example the terminal symbol "NUMBER" shown in Figure 2.5, is described using the previously defined symbol "DIGIT." The braces ("{}") are used to indicate a previously defined symbol.

```
!Lexical Section

!Lexical definitions

%define DIGIT :{0-9}
%define ALFA :{a-zA-Z}
%define ALFANUM :{a-zA-Z_0-9}

! Terminal names described using lexical definitions

NUMBER :{DIGIT}+
NAME :{ALPHA}{ALPHANUM}*

! Operator precedence

%left '+', '-'
%left '*', '/'
```

**Figure 2.5. Sample Lexical Definitions**

An important part of the lexical section is the resolution of ambiguities found in the input text that match more than one regular expression in the program. When this occurs, two rules are applied to resolve the conflict. The first rule is that when text matches more than one regular expression, the regular expression which matches the



greatest number of characters is applied. The second rule is that if the input text matches more than one token, the rule which occurs first has precedence.

Conflicts between productions in the grammar are resolved by operator precedence declarations. Operator precedence declarations begin with %left, %right, or %nonassoc. The keyword "%left" in Figure 2.5 implies left associativity for the line (e.g.,  $6 + 4 + 7$  associates as  $(6 + 4) + 7$ ). The operators with the weakest binding power are listed first. Operators on the same line have equal precedence.

#### **4. Attribute Declaration Section**

The attribute declarations section consists of attribute declarations for all non-terminals and named terminals in the program. There are two simple data types for attributes: strings and integers. The data type for each attribute of each terminal or non-terminal must be declared in this section, as in Figure 2.6.

Named terminal symbols (e.g., "AND" in Figure 2.6) are permitted two special predefined attributes that are provided to the programmer. The first is "%text," which is a string type and is initialized to the input text matched by the terminal symbol. The second is "%line," which is an integer type and is initialized to the line number of the input text on which the matching text occurred.

```

%% ! Separates lexical scanner section from attribute declarations
! section.
! Attribute declarations
.
.
.
module_header{
    name :string;
    generic_parameters :string;
};
messages{
    formal_parameters :string;
    exception_declarations :string;
};
message{
    formal_parameters :string;
    return_type :string;
    exception_declarations :string;
};
AND{
    %text :string;
};
.
.
.
%% ! Separates attribute declarations section from attribute
! grammar section.

```

**Figure 2.6. Sample Attribute Declarations**

## **5. Attribute Grammar Section**

The attribute grammar section defines the syntax and semantics of the translation. It consists of a set of Bachus-Naur Form (BNF) rules and sets of equations defining attributes. The rule syntax is shown in Figure 2.7.

```
non_terminal :    symbol-1 symbol-2 symbol-3
               {
                   ! Attribute equations go here.
               };
```

**Figure 2.7. Syntax of Kodyak Rules**

The rule in Figure 2.7 means the non-terminal can be recognized if the symbols listed after the ":" appear in the sequence shown. If the non-terminal is recognized, the attribute equations for those symbols will be evaluated.

Kodyak includes additional functions of potential value that have not been discussed here. Explanations of these functions may be found in Reference 6.

## **6. Using Kodyak**

The Kodyak input program must be stored in a file named with a ".k" suffix (e.g., program\_name.k). The command to invoke Kodyak is "k program\_name" where "program\_name.k" is the input program to be compiled. Once Kodyak has started running, it creates and then deletes several files during its processing. When it has finished processing, the input program is still present along with three other newly created files. The object code will be located in the file named "program\_name," compilation statistics are stored in "program\_name.stats," and the file "program\_name.z" will contain a symbolic listing of the parser tables and other diagnostic information.

## **D. TOOLS IMPLEMENTED WITH ATTRIBUTE GRAMMAR TOOLS**

Some previous applications of the Kodiyak Application generator are described here.

### **1. Specification Language Type Checker**

The purpose was to implement a language-dependent type checker for the specification language Spec. The code to create this tool is written entirely as an attribute grammar and then compiled using the Kodiyak Application Generator, producing the executable code [Ref. 10].

### **2. Language Translator For The Computer-Aided Prototyping System (CAPS)**

The PSDL translator translates prototype specifications written in the Prototype Description Language (PSDL) into Ada. The design of the PSDL language translator used a "template translation methodology" developed by the author [Ref. 11] that is used in the research presented here. These templates are used to derive the attribute equations that form the core of the Kodiyak source files.

### **3. Test Result Classifier**

The test result classifier is a tool developed to partially automate the testing phase of a large software development project. The test result classifier repeatedly calls a program module and reports the cases when the results of the call do not conform to the specification of the module [Ref. 12]. Spec, Kodiyak, and Ada were used in the development of the test result classifier.

### **4. Specification Pretty Printer**

A pretty printer is a software tool used to format expressions of a formal language in a consistent manner so they are easier to

understand and read. The specification pretty printer is designed to correctly format the formal specification language Spec [Ref. 13]. Kodiyak was used to generate the specification pretty printer.

### **III. DESCRIPTION OF THE DESIGN AND IMPLEMENTATION**

This chapter explains the design and implementation of the concrete interface generator. The concrete interface generation system is a software tool designed to automatically generate Ada specifications from a formal specification language.

To avoid confusion throughout this chapter, Spec keywords will be written in uppercase (e.g., MESSAGE). The curly braces "{}" will be used to denote an attribute name, such as {name}.

#### **A. THE PROBLEM**

The development of a large software system is a time-consuming, error-prone process. Most often, the design phase of the development process is rushed so the coding phase can begin. However, the design is probably the most important part of the development process. If there were a tool to automatically generate some of the actual code, more time could be spent in the design phase.

##### **1. Why Ada Was Chosen**

Ada was chosen to be the target programming language for the concrete interface generator for several reasons. One reason is the U.S. Department of Defense (DOD) directive that Ada is to be used in developing software systems for DOD [Ref. 14]. Other reasons are related to the design goals of the language itself. Ada was designed to support abstraction, information hiding, and modularity, which makes it very compatible with Spec.

## **2. Packages**

One of the basic program units in Ada is the package. Packages are used to define a collection of logically related subprogram units (procedures and/or functions), type declarations, and associated operations. A complete Ada package is divided into two parts:

1. **Specifications**—The package specification gives the compiler the information it needs for type checking, storage allocation, and generating calls for resources defined in the package. It should also give the user information about what the package does, but this is accomplished via informal comments rather than via formal Ada structures.
2. **Body**—The body contains the details of the package, such as the procedures, functions, etc. to actually implement the abstract idea described by the Spec.

Figure 3.1 shows a complete Ada package. Since packages are probably the most frequently used program units in Ada, any software tool designed to generate Ada code would be expected to generate the package construct. The package program unit is included in the current implementation of the concrete interface generator.

## **3. Limitations of this Implementation**

Only a subset of the Spec grammar has been implemented in this version of the concrete interface generator system. The implementation presented here includes Spec functions and generates Ada specifications for packages, generic functions, and non-generic functions. The current implementation does not include Spec MACHINES or TYPES, although the current design covers these cases.

<pre> generic   precision: float; package square_root_pkg is   function square_root (x: float) return float;   imaginary_square_root: exception; end square_root_pkg;  package body square_root_pkg is   function square_root (x: float) return float is     z: float := x;     tolerance: float := x * precision;   begin     if x &lt; 0.0 then       raise imaginary_square_root;     end if;     while (abs(z * z - x) &gt; tolerance) loop       -- Bound: floor(abs(z - x / z) / tolerance).       z := (z + x / z) * 0.5;     end loop;     return z;   end square_root; end square_root_pkg; </pre>	<div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin: 0 auto; width: 20px;"></div> <p>package specification</p> <div style="border-left: 1px solid black; border-right: 1px solid black; height: 250px; margin: 0 auto; width: 20px;"></div> <p>package body</p>
---	--

**Figure 3.1. Complete Ada Package**

The implementation requires that the user provide the following resources to facilitate the creation and use of the Ada specification produced by the concrete interface generation system.

1. Correct Spec input
2. Type declarations for types contained in the Spec

Correct Spec input is needed because the system does not have an error-checking capability. The user also supplies Ada packages containing the declarations of all data types and any required I/O routines for those types.



The Spec description of the generic square\_root function is shown in Figure 3.2. The Ada function "square\_root" shown in Figure 3.1 is implemented based on the Spec description of the required abstract interface, in accordance with Spec concrete interface conventions [Ref. 1]. An example of a procedure using "square\_root\_pkg" is shown in Figure 3.3. The procedure "Main" provides all the declarations and I/O routines needed by the package.

```

FUNCTION square_root(precision:float SUCH THAT precision > 0.0)
  MESSAGE(x: float)
    WHEN x >= 0.0
      REPLY(y: float)
        WHERE y > 0.0, approximates(y * y, x)
      OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: float)
    VALUE(b: boolean)
      WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END

```

**Figure 3.2. Spec Description of square\_root Function**

```

with square_root_pkg, text_io;
use text_io;
procedure main is
  package sq_rt is new square_root_pkg(float'epsilon);
  package f_io is new float_io(float);
  use sq_rt, f_io;
  s, y: float;
begin
  put("input value: ");
  get(x);
  y := square_root(x);
  put(y);
end main;

```

**Figure 3.3. Procedure Using square\_root\_pkg**

#### **4. Intended Usage**

The purpose of the concrete interface generation system is to reduce the time needed to implement a large software system after it has been formally specified with Spec. Since the Ada code is automatically generated the number of coding errors will be greatly reduced.

### **B. THE DESIGN**

#### **1. Mapping Spec to Ada**

General rules for generating concrete interface specifications in Ada corresponding to an abstract architectural design expressed in Spec were developed in the book *Software Engineering with Abstractions* by Berzins and Luqi [Ref. 5]. MESSAGE is the key construct of Spec that determines which Ada program unit or subprogram unit the Spec module will be translated into. The number of output parameters and the Spec constructs associated with the message, such as REPLY and GENERATE, greatly affect the end result. The rule that applies to the FUNCTION "square\_root" is as follows:

A message with a REPLY containing exactly one data component and without any TRANSITION clauses corresponds to an Ada function with an "in" parameter for each component of the MESSAGE and a "return" corresponding to the single data component of the REPLY. [Ref. 1]

Once the program unit has been determined, details of mapping Spec to Ada are driven by the Spec grammar. Each word of the Spec module provides information to determine the completed Ada specification.

There are two other general rules for the mapping of Spec to Ada not implemented here. These rules determine the type of Ada procedure into which the Spec will be translated. When the MESSAGE includes a Spec GENERATE, the following rule applies:

A message with a GENERATE corresponds to a generic procedure with a single generic procedure parameter. The generic parameter represents the body of the loop to be driven by the generator, and takes one "in" parameter corresponding to the elements of each sequence to be generated. The state variables of the loop correspond to nonlocal variables of the actual procedure bound to the generic parameter. [Ref. 1]

If the Spec construct does not correspond to either of the two cases described above, the following rule will apply:

A message that does not match the previous two cases corresponds to an Ada procedure with an "in" parameter for each component of the MESSAGE and an "out" parameter for each component of the REPLY, if there is one. [Ref. 1]

Additional alternatives to these rules can be introduced by Spec pragmas [Ref. 1]. These additional alternatives are not supported by the current version of the concrete interface generator.

## **2. Translation Templates**

A translation template is a schematic representation of the output desired from the translation process, which in this case is an Ada program unit. Translation templates are used for organizing and naming the attributes needed for the implementation. The attribute names identify slots to be filled by the attribute grammar. There are two steps in building the templates:

1. Determine the fixed portions of the program unit.

## 2. Determine and name the computed portions.

The fixed portions of the template consist of Ada reserved words, lexical elements, separators, and delimiters. These are shown in plain type in Figure 3.4. The portions of the program that must be determined using attribute equations are given names and analyzed. These parts should be broken down into manageable pieces and assigned attribute names which are descriptive of the Ada parts they will represent.

```
generic_parameters  -- only if required

package name_pkg is

    subprogram_declarations

    exception_declarations ;

end name_pkg;
```

**Figure 3.4. Package Template**

A common case in translation templates is a subtemplate that can be repeated, according to the structure of the source text. We express such situations via a naming convention, where the name of a slot containing repeated instances of a subtemplate is the plural form of the name of the subtemplate. The name of the subtemplate is always singular, and the associated diagram represents a single typical instance of the subtemplate. For example, the attribute {subprogram\_declarations} name is in the plural form because a package could have more than one subprogram unit (see Figure 3.4).

Another useful guideline to use when building templates is to use subtemplates when a portion of the template is not easily manageable. A template for a Spec package, with the attribute names in bold type, is shown in Figure 3.4. The function declaration template, in Figure 3.5, is a subtemplate for the package template in Figure 3.4.

```
generic_parameters  
  
function name ( input_parameters ) return return_type ;
```

Figure 3.5. **Function Declaration Template**

The translation templates for the future implementation of a Spec MACHINE are shown in Figures 3.6, 3.7, and 3.8. The procedure declaration subtemplates shown in Figure 3.7 and Figure 3.8 were introduced to show the parameter structure and attribute names of an Ada procedure.

A translation template for a Spec TYPE is given in Figure 3.9. The TYPE template requires the additional attribute {type\_declarations} to build the parts needed in Ada for coding abstract data types. The generated type declaration in the private part will have an empty slot for the actual definition of the data representation which must be filled in during implementation. This information can also be derived from a Spec pragma in a future version of the system.

The template process is very useful and should be used throughout the implementation.

```

generic_parameters

package name_pkg is

    subprogram_declarations

    exception_declarations

end name_pkg;

```

**Figure 3.6. Machine Declaration Template**

```

generic_parameters
procedure name ( procedure_parameters ) ;

```

**Figure 3.7. Procedure Declaration Template**

```

parameter_name : parameter_mode parameter_type

```

**Figure 3.8. Procedure Parameters Template**

```

generic_parameters

package name_pkg is

    type_declaration

    subprogram_declarations

    exception_declarations

    private
        type_declaration

end name_pkg;

```

**Figure 3.9. Type Template**

## C. IMPLEMENTATION

The purpose of attributes used in the translation is to construct the computed parts of the desired Ada specification. The attribute names were chosen based on the translation templates.

The reader should refer to Figures 3.4, 3.5, 3.10, and 3.11 in order to understand the relationship between the various tools used in the design and clarify any questions that may arise from the following discussion.

A sample production rule for the non-terminal "message" with sample attribute equations is shown below in Figure 3.10. The attribute names are in bold type and show how attribute values are passed up the parse tree. The attribute dependency diagram shown in Figure 3.11 summarizes dependencies between the attributes and is discussed in Part 2 of Section C.

```
message

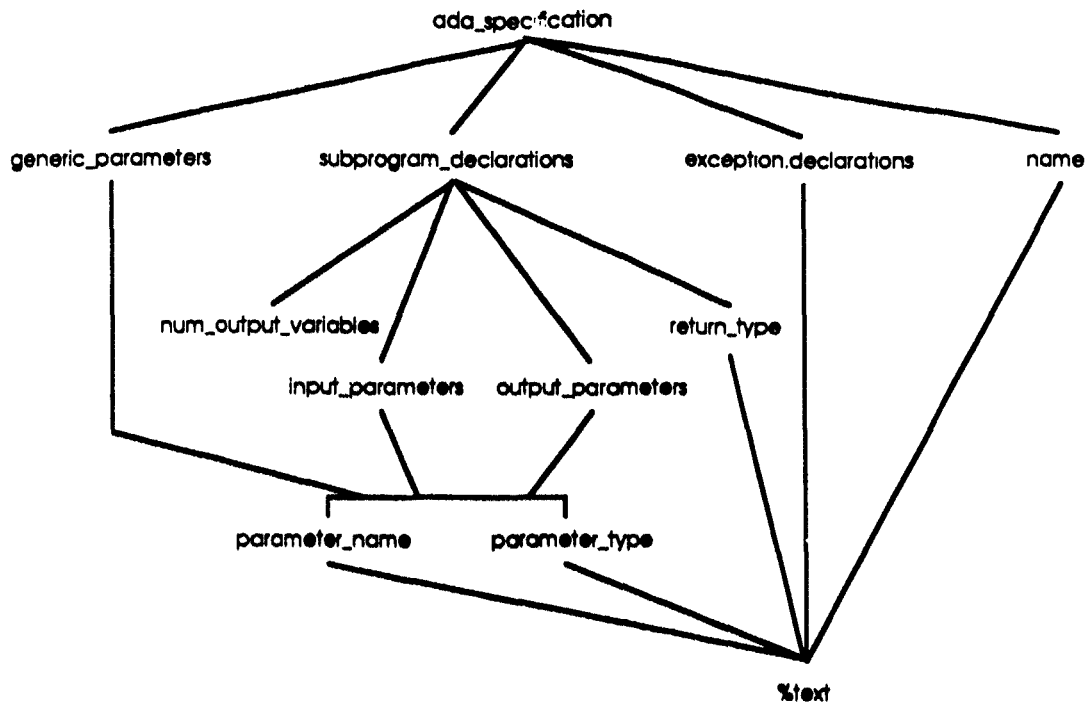
: MESSAGE formal_message response pragmas

{ message.formal_parameters =
    formal_message.formal_parameters;

  message.return_type = response.return_type;

  message.exception_declarations =
    response.exception_declarations;
```

Figure 3.10. Production Rule With Attributes



**Figure 3.11. Attribute Dependency Diagram**

### **1. Attribute Characteristics**

There are two characteristics used to describe an attribute. One characteristic describes how the attribute is derived (synthesized or inherited) and the other characteristic describes the attribute's data type.

An attribute is either synthesized or inherited. For synthesized attributes, the attribute value of the symbol on the left-hand side of a production is defined in terms of the attributes of the symbols on the right-hand side. Synthesized attributes are computed bottom-up. Conversely, inherited attributes for symbols on the right are defined in terms of the attributes of the symbol on the left-hand side. Inherited attributes are computed top-down.



The type of an attribute refers to the data type associated with the values of the attribute. The only types allowed by Kodiyak are integer, string, and map.

For the design presented here, all attributes are synthesized and have values of type string except for the attribute {num\_output\_variables}, which is type integer.

## **2. Attribute Dependency Diagram**

The attribute dependency diagram is another useful tool for understanding a design based on an attribute grammar. The diagram shown in Figure 3.11, should be constructed during the implementation process. The dependency diagram's purpose is similar to that for a module dependency diagram for a program and shows which parts are needed to compute the value of each attribute.

Several pieces of information are determined from analyzing the dependency diagram. For example, the attribute {ada\_specification} requires the values of every other attribute before its value can be determined. The diagram also shows which attributes are used more than once in the implementation. The diagram should prove extremely valuable for extending or changing the implementation at a later date.

## **3. Attribute Definitions**

The attribute dependency diagram (Figure 3.11), the package template (Figure 3.4), and the function declarations template (Figure 3.5) summarize the structure of the design. Each attribute is connected to a set of attributes on the next lower level of the diagram. This is the set of attributes used to define the parent attribute. For example, the attribute

{ada\_specification} is defined in terms of the attributes {generic\_parameters}, {name}, {subprogram\_declarations}, and {exception\_declarations}. The following discussion is organized in top-down order with respect to the attribute dependency diagram.

**a. *ada\_specification***

The highest level attribute {ada\_specification} is used to produce the translation output. All of the other attributes are built, concatenated together, and stored in the attribute {ada\_specification}. The value of the {ada\_specification} attribute at the root node of the parse tree provides the result of the translation, as indicated by a "%output" declaration.

**b. *generic\_parameters***

The attribute {generic\_parameters} is used to build the generic portion of the package if there is one. When a package contains generic parameters, this attribute provides the Ada reserved words, variable names, type, and delimiters needed for the generic part of an Ada specification.

**c. *name***

The attribute {name} represents the actual name of the package, which is "square\_root" for the example shown in Figure 3.2. As shown in the package template in Figure 3.4, the value of an attribute can be used more than once. The attributes found on the same level as {name} are the ones used to build the template.

#### **d. *subprogram\_declarations***

The attribute {subprogram\_declarations} is used to build the required subprogram declaration statements. It uses the attribute {parameter\_name} and {parameter\_type} to build the complete parameter list. This attribute also provides the necessary keywords for the subprogram declaration statement.

#### **e. *exception\_declarations***

The last attribute used in the package template is {exception\_declarations}. This attribute adds the exception declaration statements to the Ada specifications that were found in the MESSAGE.

#### **f. *input\_parameters***

The attribute {input\_parameters} provides the parameter names, modes, types, and delimiters for the input parameter part of the subprogram declaration, for any function or procedure that has input parameters. The attribute {input\_parameters} and {return\_type} are shown in the function declaration template (Figure 3.5).

#### **g. *return\_type***

The attribute {return\_type} is used to furnish the type of the function's return variable. An Ada function can return only one type per Ada function and this type may or may not be the same as the type of one of the input variables.

#### **h. *output\_parameters***

The attribute {output\_parameters} provides the parameter names, mode, type, and delimiters of the output parameter list for an Ada procedure. This attribute uses {parameter\_name} and

{parameter\_type} to build the parameter list. The attribute {input\_parameters} and {output\_parameters} are concatenated together to form the procedure's complete parameter list.

**i. num\_output\_variables**

The attribute {num\_output\_variables} is used to count the number of output variables in a Spec MESSAGE. This is the only attribute of type integer in the implementation. The number of output variables is important because an Ada function can return only one value. After the output variables have been counted, the result is stored in {num\_output\_variable} and used in a decision statement to determine which subprogram declaration will be produced.

**j. parameter\_name**

The {parameter\_name} attribute's purpose is to build the parameter list for the three higher-level attributes {generic\_parameters}, {input\_parameters}, and {output\_parameters}. The {parameter\_name} attribute is not shown in the package or function declaration template because it is a lower-level attribute.

**k. parameter\_type**

The attribute {parameter\_type} is used to provide the data type for all parameters in the generic, formal, and actual parameter lists. Also a lower-level attribute, {parameter\_type} does not appear in the translation templates.

**l. %text**

The attribute {%text} is the lowest-level attribute and is pre-defined by Kodiyak. The value of this attribute is the source text that

matches a terminal symbol in the grammar. Every other attribute depends on (%text) for pieces of actual text needed from the initial Spec specification to be used in the output Ada specification. For example, the names of packages, variables, and data types are needed to produce the Ada specification.

#### **D. SYSTEM GENERATION PROCEDURE**

Once the attribute grammar has been written, it must be compiled using Kodiyak. If compilation is successful, an executable "C" program is produced. It is this executable "C" program which accepts the Spec function as input to produce the Ada specification. A more detailed procedure for actually using this implementation is explained in the user's guide found in Appendix C.

#### **E. SAMPLE INPUT AND OUTPUT**

Several samples of Spec specifications used as input for the concrete interface generation system and the respective Ada specifications generated by the system are shown below. The examples include cases of non-generic, generic, multiple input values, multiple output values, exception declarations, and multiple exception declarations, and multiple messages are demonstrated.

##### **1. Non-Generic Example**

The first example in Figure 3.12 shows a non-generic Spec FUNCTION and the corresponding Ada specification generated as output. The generated Ada package contains one function statement resulting from the MESSAGE and one exception declaration resulting from the

OTHERWISE REPLY EXCEPTION statement. The Spec construct CONCEPT does not contribute to the Ada specification but does provide information for the package body, as well as for the test result evaluator described in Reference 12.

```

FUNCTION square_root
  MESSAGE(x: float)
    WHEN x >= 0.0
      REPLY(y: float)
        WHERE y > 0.0, approximates(y * y, x)
      OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: float)
    VALUE(b: boolean)
      WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END

```

#### A. Input: Spec

```

package square_root_pkg is
  function name(x: in float) return float;
  imaginary_square_root: exception;
end square_root_pkg;

```

#### B. Output: Ada Specification

**Figure 3.12. Non-Generic Example**

## 2. Generic Example

A generic FUNCTION was used for the example shown in Figure 3.13. In this example, the variable precision is generic, and the generic parameter declaration in the Ada specification is generated from the Spec

generic parameter declaration. The rest of the Ada specification is similar to the non-generic example above.

```
FUNCTION square_root(precision:float SUCH THAT precision > 0.0
  MESSAGE(x: float)
    WHEN x >= 0.0
      REPLY(y: float)
        WHERE y > 0.0, approximates(y * y, x)
      OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: float)
    VALUE(b: boolean)
      WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END
```

#### A. Input: Spec

```
generic
  precision: float;
package square_root_pkg is
  function name(x: in float) return float;
  imaginary_square_root: exception;
end square_root_pkg;
```

#### B. Output: Ada Specification

Figure 3.13. Generic Example

### 3. Multiple—Messages, Inputs

The example Spec FUNCTION “plus” shown in Figure 3.14 demonstrates a very important feature of the concrete interface generation system. A Spec FUNCTION can contain more than one MESSAGE and must generate a function declaration for each instance. This is useful for grouping together the overloaded variants of a function, especially

if the meanings of the variants are strongly related. This example also demonstrates the capability of the concrete interface generation system to generate the Ada specification for Spec inputs containing multiple parameter inputs.

```
FUNCTION plus
  MESSAGE(x:integer, y:integer)
    REPLY(z:integer)
    WHERE z = x + y

  MESSAGE(x:real, y:integer)
    REPLY(z:real)
    WHERE z = x + y

  MESSAGE(x:integer, y:real)
    REPLY(z:real)
    WHERE z = integer_to_real(x) + y
END
```

#### A. Input: Spec

```
package plus_pkg is
  function name(x: in integer; y: in integer) return integer;
  function name(x: in real; y: in integer) return real;
  function name(x: in integer; y: in real) return real;
end plus_pkg;
```

#### B. Output: Ada Specification

**Figure 3.14. Example of Multiple Messages and Multiple Inputs**



#### 4. Multiple Outputs, Exception Declarations

The example in Figure 3.15 is used to demonstrate the case of a Spec FUNCTION containing more than one output. If a Spec FUNCTION contains more than one output, an Ada procedure statement is required.

```
FUNCTION divide
  MESSAGE(x: integer, y:integer)
    WHEN y /= 0
      REPLY(quotient:integer, remainder:integer)
    WHEN X=0, Y=0
      REPLY EXCEPTION result_indeterminate
    OTHERWISE REPLY EXCEPTION result_infinite
END
```

##### A. Input: Spec

```
package divide_pkg is
  procedure name(x: in integer; y: in integer;
    quotient: out integer; remainder: out integer);
  result_infinite: exception;
  result_indeterminate: exception;
end divide_pkg;
```

##### B. Output: Ada Specification

**Figure 3.15. Example of Multiple Outputs and Multiple Exception Declarations**

This example includes multiple exception declaration statements. As a formatting feature, when there are multiple messages, the system was designed to add all exception declarations at the end of the Ada specification as a group.

## **IV. CONCLUSIONS**

This thesis has shown the design and implementation of a concrete interface generation system is feasible. With the aid of the concrete interface generation system, an Ada specification for the implemented subset of Spec can be automatically generated in less time than it would take a person to enter a hand-coded version into the computer. A fully implemented concrete interface generation system using Spec as the formal specification language will save many man-hours and greatly reduce the error rate during the implementation phase of a software development project.

### **A. EVALUATION OF THE DESIGN**

The use of translation templates and attribute dependency diagrams proved to be very useful tools in designing the concrete interface generator. To ensure success, the implementation had to be approached in a systematic fashion. The translation templates were very useful for the organization and naming of attributes before the implementation began. The attribute dependency diagram contributed to the design, during the implementation process, by serving as a quick reference for attribute dependency. Translation templates and attribute dependency diagram are both highly recommended for future extensions of the concrete interface generation system.

## **B. EVALUATION OF THE IMPLEMENTATION**

The greatest impact on the implementation effort was due to Kodiyak's lack of features and lack of user-friendliness. The implementation process could have been much faster if Kodiyak had easily understood error messages, better debugging facilities, and a good user interface.

The development of a good user interface could greatly enhance the use of a tool like Kodiyak. For example, a graphical interface with the ability to select attribute names from an attribute menu, display the grammar in tree form, and move about the tree easily by selecting the node to be displayed could greatly reduce the time required to implement a software tool.

Due to Kodiyak's limited data types and lack of options to define types or constants, several features could be added to create a more robust environment for the generation of software tools. A few features that would be helpful include the ability to define symbolic constants, create user defined functions easily, implement user defined data types, and interface with a programming language like Ada.

## **C. FUTURE WORK**

This thesis has shown the feasibility of creating and implementing a concrete interface generation system. However, more work must be done to implement the complete Spec language. The implementation of Spec MACHINE and TYPE modules plus the implementation of pragmas for MACHINE, FUNCTION, and TYPE modules will be needed for a complete implementation of the concrete interface generation system.

## APPENDIX A

### GRAMMAR FOR THE FORMAL LANGUAGE SPEC

This appendix provides a listing of the Spec grammar used in the implementation of the concrete interface generation system.

```
start
    : spec
    ;

spec
    : spec module
    |
    ;
    ! A production with nothing after the "|" means the empty string
    ! is a legal replacement for the left hand side.

module
    : definition
    | function
    | type
    | machine
    | instance ! of a generic module
    ;

function
    : optionally_virtual FUNCTION module_header messages concepts END
    ;
    ! Virtual modules are for inheritance only, never used directly.

machine
    : optionally_virtual MACHINE module_header state messages
    temporals transactions concepts END
    ;

type
    : optionally_virtual TYPE module_header model messages temporals
    transactions concepts END
    ;
```

```

definition
    : DEFINITION module_header concepts END
    ;

instance
    : INSTANCE module_header where foreach concepts END
    ;
    ! For making instances or partial instantiations of generic
modules.
    ! The foreach clause allows defining sets of instances.

module_header
    : formal_name defaults inherits imports export pragmas
    ;
    ! This part describes the static aspects of a module's
interface.
    ! The dynamic aspects of the interface are described in the
messages.
    ! A module is generic iff it has parameters.
    ! The parameters can be constrained by a SUCH THAT clause.
    ! A module can inherit the behavior of other modules.
    ! A module can import concepts from other modules.
    ! A module can export concepts for use by other modules.

pragmas
    : pragmas PRAGMA actual_name '(' actuals ')'
    |
    ;

inherits
    : inherits INHERIT actual_name hide renames
    |
    ;
    ! Ancestors are generalizations or simplified views of a module.
    ! A module inherits all of the behavior of its ancestors.
    ! Hiding a message or concept means it will not be inherited.
    ! Inherited components can be renamed to avoid naming conflicts.

```

```

hide
    : HIDE name_list
    |
    ;
    ! Useful for providing limited views of an actor.
    ! Different user classes may see different views of a system.
    ! Messages and concepts can be hidden.

renames
    : renames RENAME NAME AS NAME
    |
    ;
    ! Renaming is useful for preventing NAME conflicts when
inheriting
    ! from multiple sources, and for adapting modules for new uses.
    ! The parameters, model and state components, messages,
exceptions,
    ! and concepts of an actor can be renamed.

imports
    : imports IMPORT name_list FROM actual_name
    |
    ;

export
    : EXPORT name_list
    |
    ;

messages
    : messages message
    |
    ;

message
    : MESSAGE formal_message pragmas response
    ;

response
    : response_set
    | response_cases
    ;

```

```

response_cases
    : WHEN expression_list response_set pragmas response_cases
    | OTHERWISE response_set pragmas
    ;

response_set
    : choose reply sends transition
    ;

choose
    : CHOOSE '(' formals ')'
    |
    ;

reply
    : REPLY actual_message where
    | GENERATE actual_message where      ! used in generators
    |
    ;

sends
    : sends send
    |
    ;

send
    : SEND actual_message TO actual_name where foreach
    ;

transition
    : TRANSITION expression_list ! for describing state changes
    |
    ;

formal_message
    : optional_exception optional_formal_name formal_arguments
defaults
    ;

actual_message
    : optional_exception optional_actual_name formal_arguments
    ;

```

```

defaults
    : DEFAULT expression_list
    |      %prec SEMI ! must have a lower precedence than DEFAULT
    ;

where
    : WHERE expression_list
    |      %prec SEMI ! must have a low : precedence than WHERE
    ;

optionally_virtual
    : VIRTUAL
    |
    ;

optional_exception
    : EXCEPTION
    |      %prec SEMI
    ;

foreach
    : FOREACH '(' formals ')'
    |
    ;
    ! foreach is used to describe a set of messages or instances

model      ! data types have conceptual models for values
    : MODEL formal_arguments invariant pragmas
    |
    ;

state      ! machines have conceptual models for states
    : STATE formal_arguments invariant initially pragmas
    |
    ;

invariant      ! invariants are true for all states or instances
    : INVARIANT expression_list
    ;

initially      ! initial conditions are true only at the beginning
    : INITIALLY expression_list
    ;

```



temporals

```
: temporals temporal
|
;
```

temporal

```
: TEMPORAL formal_name defaults where response
;
! Temporal events are triggered at absolute times,
! in terms of the local clock of the actor.
! The "where" describes the triggering conditions
! in terms of TIME, PERIOD, and DELAY.
```

transactions

```
: transactions transaction
|
;
```

transaction

```
: TRANSACTION actual_name '=' action_list where foreach
;
! Transactions are atomic.
! The where clause can specify timing constraints.
```

action\_list

```
: action_list ';' action %prec SEMI ! sequence
| action
;
```

action

```
: action action %prec STAR ! unordered set of actions
| IF alternatives FI ! choice
| DO alternatives OD ! repeated choice
| actual_name ! a normal message or subtransaction
| EXCEPTION actual_name ! an exception message
;
```

alternatives

```
: alternatives OR guard action_list
| guard action_list
;
```

```

guard
    : WHEN expression_list ARROW
    |
    ;

concepts
    : concepts concept
    |
    ;

concept
    : CONCEPT formal_name ':' expression defaults where
      ! constants
    | CONCEPT for _name '(' formals ')' defaults VALUE '(' formals
      ')' where
      ! func ons, defined with preconditions and postconditions
    ;

optional_formal_name
    : formal_name
    |
    ;

formal_name
    : NAME '(' formals ')'
    | NAME
    ;

formal_arguments
    : '(' formals ')'
    |
    ;

formals
    : field_list restriction
    ;

field_list
    : field_list ',' type_binding
    | type_binding
    ;

```

```

type_binding
    : name_list ':' expression
    | '$' NAME ':' expression
    ;

name_list
    : name_list NAME
    | NAME
    ;

restriction
    : SUCH expression_list
    ;

optional_actual_name
    : actual_name
    ;

actual_name
    : NAME '(' actuals ')'
    | NAME %prec SEMI ! must have a lower precedence than '('
    ;

actuals
    : actuals ',' arg %prec COMMA
    | arg
    ;

arg
    : expression
    | pair
    ;

expression_list
    : expression_list ',' expression %prec COMMA
    | expression %prec COMMA
    ;

```

# expression

```

: QUANTIFIER '(' formals BIND expression ')'
| actual_name      ! variables and constants
| expression '(' actuals ')' ! function call
| expression '@' actual_name ! expression with explicit type cast
| NOT expression %prec NOT
| expression AND expression %prec AND
| expression OR expression %prec OR
| expression IMPLIES expression %prec IMPLIES
| expression IFF expression %prec IFF
| expression '=' expression %prec LE
| expression '<' expression %prec LE
| expression '>' expression %prec LE
| expression LE expression %prec LE
| expression GE expression %prec LE
| expression NE expression %prec LE
| expression NLT expression %prec LE
| expression NGT expression %prec LE
| expression NLE expression %prec LE
| expression NGE expression %prec LE
| expression EQV expression %prec LE
| expression NEQV expression %prec LE
| '-' expression %prec UMINUS
| expression '+' expression %prec PLUS
| expression '-' expression %prec MINUS
| expression '*' expression %prec MUL
| expression '/' expression %prec DIV
| expression MOD expression %prec MOD
| expression EXP expression %prec EXP
| expression U expression %prec U
| expression APPEND expression %prec APPEND
| expression IN expression %prec IN
| '*' expression %prec STAR
  ! *x is the value of x in the previous state
| '$' expression %prec DOT
  ! $x represents a collection of items rather than just one
  ! s1 = (x, $s2) means s1 = union({x}, s2)
  ! s1 = [x, $s2] means s1 = append([x], s2)
| expression RANGE expression %prec RANGE
  ! x in [a .. b] iff x in (a .. b) iff a <= x <= b, [a .. b] is
sorted in increasing order
| expression '.' NAME %prec DOT
| expression '[' expression ']' %prec DOT

```

```

| '(' expression ')'
| '(' expression NAME ')'      ! expression with units of
measurement
    ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS MINUTES
HOURS DAYS WEEKS
| TIME          ! The current local time, used in temporal
events
| DELAY         ! The time between the triggering event and the
response
| PERIOD       ! The time between successive events of this type
| literal      ! literal with optional type id
| '?'         ! An undefined value to be specified later
| '!'         ! An undefined and illegal value
| IF expression THEN expression middle_cases ELSE expression FI
;

```

```

middle_cases
: middle_cases ELSE_IF expression THEN expression
|
;

```

```

literal
: INTEGER_LITERAL
| REAL_LITERAL
| CHAR_LITERAL
| STRING_LITERAL
| '#' NAME          ! enumeration type literal
| '[' expressions ']' ! sequence literal
| '(' expressions ')' ! set literal
| '(' formals BIND expression ')' ! set literal
| '(' expressions ';' expression ')' ! map literal
| '[' pair_list ']' ! tuple literal
| '(' NAME BIND expression ')' ! union literal
;
! relation literals are sets of tuples

```

```

expressions
: expression_list
|
;

```

```
pair_list
: pair_list ',' pair
| pair
;

pair
: name_list BIND expression
;
```

## APPENDIX B

### KODIYAK SOURCE CODE TO CREATE THE CONCRETE INTERFACE GENERATION SYSTEM—ADACI

```
! version stamp $Header: src.k,v 1.1 90/11/03 08:54:59 Rachal Exp $

! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%define      Digit      :{0-9}
%define      Int        :{Digit}+
%define      Lower      :{a-z}
%define      Upper      :{A-Z}
%define      Letter      :(({Lower})|({Upper}))
%define      Alpha      :(({Lower})|({Digit})|"_")
%define      Blank      :{ \t\n}
%define      Quote      :{"}
%define      Backslash  :{"\\}
%define      Char        :([^\\"|{Backslash}{Quote}|{Backslash}{Backslash})
%define      Op1         :({"&"|"~"|"="|"<="|">=")
%define      Op2         :("<"|">"|"="|"<="|">=")
%define      Op3         :("~="|"~<"|"~>"|"~<="|"~>="|"=="|"~==")
%define      Op4         :({"+"|"-"|"*"|"\/"|{Backslash}|MOD|"^")
%define      Op5         :({U|IN|"."|"|"|"."|"|"[")
%define      Op          :(({Op1})|({Op2})|({Op3})|({Op4})|({Op5}))

! definitions of white space and comments

:({Blank})+
;"--".**\n"

! definitions of compound symbols and keywords
```

AND	: "&"
OR	: " "
NOT	: "~"
IMPLIES	: ">"
IFF	: "<=>"
LE	: "<="
GE	: ">="
NE	: "~="
NLT	: "~<"
NGT	: "~>"
NLE	: "~<="
NGE	: "~>="
EQV	: "=="
NEQV	: "~=="
RANGE	: ".."
APPEND	: "  "
MOD	: {Backslash} MOD
EXP	: "^"
BIND	: ":"
ARROW	: "->"
IF	: IF
THEN	: THEN
ELSE	: ELSE
IN	: IN
U	: U
SUCH	: SUCH(Blank)*THAT
ELSE_IF	: ELSE(Blank)*IF
AS	: AS
CHOOSE	: CHOOSE
CONCEPT	: CONCEPT
DEFAULT	: DEFAULT
DEFINITION	: DEFINITION
DELAY	: DELAY
DO	: DO
END	: END
EXCEPTION	: EXCEPTION
EXPORT	: EXPORT
FI	: FI
FOREACH	: FOREACH
FROM	: FROM
FUNCTION	: FUNCTION



GENERATE	:GENERATE
HIDE	:HIDE
IMPORT	:IMPORT
INHERIT	:INHERIT
INITIALLY	:INITIALLY
INSTANCE	:INSTANCE
INVARIANT	:INVARIANT
MACHINE	:MACHINE
MESSAGE	:MESSAGE
MODEL	:MODEL
OD	:OD
OF	:OF
OTHERWISE	:OTHERWISE
PRAGMA	:PRAGMA
PERIOD	:PERIOD
RENAME	:RENAME
REPLY	:REPLY
SEND	:SEND
STATE	:STATE
TEMPORAL	:TEMPORAL
TIME	:TIME
TO	:TO
TRANSACTION	:TRANSACTION
TRANSITION	:TRANSITION
TYPE	:TYPE
VALUE	:VALUE
VIRTUAL	:VIRTUAL
WHEN	:WHEN
WHERE	:WHERE
QUANTIFIER	:{Upper}{Upper}+
NAME	:(((Letter){Alpha}*) (({Quote}{Op}{Quote})))
INTEGER_LITERAL	:{Int}
REAL_LITERAL	:{Int}" Cant {Int}
CHAR_LITERAL	: Cant
STRING_LITERAL	:{Quote}{Char}*{Quote}

! operator precedences, %left means 2+3+4 is (2+3)+4.

%left	;', IF, DO, EXCEPTION, NAME, SEMI;
%left	',' COMMA;
%left	SUCH;
%left	'@';
%left	IFF;
%left	IMPLIES;
%left	OR;
%left	AND;

```

%left      NOT;
%left      '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV,
NEQV;
%left      IN, RANGE;
%left      U, APPEND;
%left      '+', '-', PLUS, MINUS;
%left      '*', '/', MUL, DIV, MOD;
%left      UMINUS;
%left      EXP;
%left      '$', '[', '(', '{', '.', DOT, WHERE;
%left      STAR;

```

```

%%

```

```

!Explanations of attributes.

```

```

!ada_specification : synthesized string - the result of the translation.

```

```

!generic_parameters : synthesized string - builds generic portion of an
!                      Ada specification.

```

```

!name : synthesized string - provides name of the Spec module.

```

```

!subprogram_declarations : synthesized string - builds subprogram declaration
!                      statements for functions or procedures.

```

```

!exception_declarations : synthesized string - builds exception declarations
!                      portion of an Ada package.

```

```

!input_parameters : synthesized string - builds list of input parameters for
!                      the function declaration statements.

```

```

!return_type : synthesized string - provides return type of the function.

```

```

!parameter_name : synthesized string - provides parameter names.

```

```

!parameter_type : synthesized string - provides data type for variables in
!                      the parameter list.

```

```

!output_parameters : synthesized string - builds list of output parameters
!                      for the procedure declaration statements.

```

```

!num_output_variables : synthesized integer - provides an integer value
!                      used by the decision statement in the message
!                      production to decide which subprogram declaration
!                      statement is required.

```

```

!%text : synthesized string - a Kodiyak pre-defined attribute initialized to
!       the text the terminal symbol matched.

!Attribute declarations.
start( ada_specification:string; );

spec( ada_specification:string; );

module(ada_specification:string; );

function( ada_specification:string;
          generic_parameters:string;
          subprogram_declarations:string;
          exception_declarations:string;
          name:string; );

module_header( name:string;
              generic_parameters:string; );

messages( input_parameters:string;
          return_type:string;
          subprogram_declarations:string;
          exception_declarations:string; );

message( input_parameters:string;
         return_type:string;
         subprogram_declarations:string;
         output_parameters:string;
         num_output_variables:int;
         exception_declarations:string; );

response( return_type:string;
         output_parameters:string;
         num_output_variables:int;
         exception_declarations:string; );

response_cases( return_type:string;
               output_parameters:string;
               num_output_variables:int;
               exception_declarations:string; );

response_set( return_type:string;
             output_parameters:string;
             num_output_variables:int;
             exception_declarations:string; );

```

```

reply( return_type:string;
       output_parameters:string;
       num_output_variables:int;
       exception_declarations:string; );

formal_message( input_parameters:string; );

actual_message( return_type:string;
                output_parameters:string;
                num_output_variables:int;
                exception_declarations:string; );

formal_name( name:string;
             generic_parameters:string;
             %text:string; );

formal_arguments( input_parameters:string;
                  output_parameters:string;
                  num_output_variables:int;
                  return_type:string; );

formals( input_parameters:string;
          output_parameters:string;
          generic_parameters:string;
          num_output_variables:int;
          return_type:string; );

field_list( input_parameters:string;
            output_parameters:string;
            generic_parameters:string;
            num_output_variables:int;
            return_type:string; );

type_binding( input_parameters:string;
              output_parameters:string;
              generic_parameters:string;
              parameter_name:string;
              parameter_type:string;
              return_type:string; );

name_list( parameter_name:string;
           %text:string; );

optional_actual_name( exception_declarations:string; );

```

```

actual_name( parameter_type:string;
              return_type:string;
              exception_declarations:string;
              %text:string; );

expression( parameter_type:string;
             return_type:string; );

EXCEPTION( %text:string; );

QUANTIFIER( %text:string; );

NAME( %text:string; );

INTEGER_LITERAL( %text:string; );

REAL_LITERAL( %text:string; );

CHAR_LITERAL( %text:string; );

STRING_LITERAL( %text:string; );

%%
! productions of the grammar

start
    : spec
      ( %output(spec.ada_specification); )
    ;

spec
    : spec module
      { spec[1].ada_specification =
        [spec[2].ada_specification,module.ada_specification;; ]
      |
        { spec.ada_specification = ""; }
      ;
      ! A production with nothing after the "|" means the empty string
      ! is a legal replacement for the left hand side.

module
    : definition
      { }
    | function
      { module.ada_specification = function.ada_specification; }
    | type
      { }

```

```

| machine
  { }
| instance    ! of a generic module
  { }
;

function
: optionally_virtual FUNCTION module_header messages concepts END
  { function.ada_specification =
    [module_header.generic_parameters,
     "package ", module_header.name, "_pkg is\n",
     messages.subprogram_declarations,
     messages.exception_declarations,
     "end ", module_header.name, "_pkg;\n\n"]; }
;
! Virtual modules are for inheritance only, never used directly.

machine
: optionally_virtual MACHINE module_header state messages temporals
  transactions concepts END
  { }
;

type
: optionally_virtual TYPE module_header model messages temporals
  transactions concepts END
  { }
;

definition
: DEFINITION module_header concepts END
  { }
;

instance
: INSTANCE module_header where foreach concepts END
  { }
;
! For making instances or partial instantiations of generic modules.
! The foreach clause allows defining sets of instances.

module_header
: formal_name defaults inherits imports export pragmas
  { module_header.name = formal_name.name;
    module_header.generic_parameters = formal_name.generic_parameters; }
;

```

```

! This part describes the static aspects of a module's interface.
! The dynamic aspects of the interface are described in the messages.
! A module is generic iff it has_parameters.
! The_parameters can be constrained by a SUCH THAT clause.
! A module can inherit the behavior of other modules.
! A module can import concepts from other modules.
! A module can export concepts for use by other modules.

```

pragmas

```

: pragmas PRAGMA actual_name '(' actuals ')'
( )
|
( )
;

```

inherits

```

: inherits INHERIT actual_name hide renames
( )
|
( )
;

! Ancestors are generalizations or simplified views of a module.
! A module inherits all of the behavior of its ancestors.
! Hiding a message or concept means it will not be inherited.
! Inherited components can be renamed to avoid naming conflicts.

```

hide

```

: HIDE name_list
( )
|
( )
;

! Useful for providing limited views of an actor.
! Different user classes may see different views of a system.
! Messages and concepts can be hidden.

```

renames

```

: renames RENAME NAME AS NAME
( )
|
( )
;

! Renaming is useful for preventing NAME conflicts when inheriting
! from multiple sources, and for adapting modules for new uses.
! The_parameters, model and state components, messages, exceptions,
! and concepts of an actor can be renamed.

```

```

imports
: imports IMPORT name_list FROM actual_name
  ( )
|
  ( )
;

export
: EXPORT name_list
  ( )
|
  ( )
;

messages
: messages message
  ( messages[1].subprogram_declarations =
    [messages[2].subprogram_declarations,
    message.subprogram_declarations];
    messages[1].exception_declarations =
    [messages[2].exception_declarations,
    message.exception_declarations]; )
|
  ( messages.subprogram_declarations = "";
    messages.exception_declarations = ""; )
;

message
: MESSAGE formal_message response pragmas
  ( message.subprogram_declarations =
    response.num_output_variables == 1
    -> ["\tfunction name(", formal_message.input_parameters,
      ") return ", response.return_type, ";\n"]
    # ["\tprocedure name(", formal_message.input_parameters,
      ";\n\t\t\t", response.output_parameters, ");\n"];
    message.exception_declarations = response.exception_declarations; )
;

response
: response_set
  ( response.return_type = response_set.return_type;
    response.output_parameters = response_set.output_parameters;
    response.num_output_variables = response_set.num_output_variables;
    response.exception_declarations = ""; )
| response_cases
  ( response.return_type = response_cases.return_type;
    response.output_parameters =

```



```

        response_cases.output_parameters;
response.num_output_variables =
        response_cases.num_output_variables;
response.exception_declarations =
        response_cases.exception_declarations; }

;

response_cases
: WHEN expression_list response_set pragmas response_cases
  { response_cases[1].return_type =
    [response_cases[2].return_type, response_set.return_type];
    response_cases[1].exception_declarations =
    [response_cases[2].exception_declarations,
    response_set.exception_declarations];
    response_cases[1].output_parameters =
    [response_cases[2].output_parameters,
    response_set.output_parameters];
    response_cases[1].num_output_variables =
    response_set.num_output_variables; }
| OTHERWISE response_set
  { response_cases[1].exception_declarations =
    response_set.exception_declarations;
    response_cases.num_output_variables =
    response_set.num_output_variables;
    response_cases.return_type = "";
    response_cases.output_parameters = ""; }

;

response_set
: choose reply sends transition
  { response_set.return_type = reply.return_type;
    response_set.output_parameters = reply.output_parameters;
    response_set.num_output_variables = reply.num_output_variables;
    response_set.exception_declarations =
    reply.exception_declarations; }

;

choose
: CHOOSE '(' formals ')'
  { }

;

reply
: REPLY actual_message where
  { reply.return_type = actual_message.return_type;
    reply.output_parameters = actual_message.output_parameters;

```

```

        reply.num_output_variables = actual_message.num_output_variables;
        reply.exception_declarations =
            actual_message.exception_declarations; }
| GENERATE actual_message where          ! used in generators
  { }
|
  { reply.return_type = "";
    reply.output_parameters = "";
    reply.exception_declarations = ""; }
;

sends
: sends send
  { }
|
  { }
;

send
: SEND actual_message TO actual_name where foreach
  { }
;

transition
: TRANSITION expression_list          ! for describing state changes
  { }
|
  { }
;

formal_message
: optional_exception optional_formal_name formal_arguments defaults
  { formal_message.input_parameters =
    formal_arguments.input_parameters; }
;

actual_message
: optional_exception optional_actual_name formal_arguments
  { actual_message.return_type = formal_arguments.return_type;
    actual_message.output_parameters =
      formal_arguments.output_parameters;
    actual_message.num_output_variables =
      formal_arguments.num_output_variables;
    actual_message.exception_declarations =
      optional_actual_name.exception_declarations; }
;

```

```

defaults
: DEFAULT expression_list
( )
|      %prec SEMI      ! must have a lower precedence than DEFAULT
( )
;

where
: WHERE expression_list
( )
|      %prec SEMI      ! must have a lower precedence than WHERE
( )
;

optionally_virtual
: VIRTUAL
( )
|
( )
;

optional_exception
: EXCEPTION
( )
|      %prec SEMI
( )
;

foreach
: FOREACH '(' formals ')'
( )
|
( )
;
! foreach is used to describe a set of messages or instances

model      ! data types have conceptual models for values
: MODEL formal_arguments invariant pragmas
( )
|
( )
;

```

```

state      ! machines have conceptual models for states
: STATE formal_arguments invariant initially pragmas
  { }
  |
  { }
;

invariant   ! invariants are true for all states or instances
: INVARIANT expression_list
  { }
;

initially   ! initial conditions are true only at the beginning
: INITIALLY expression_list
  { }
;

temporals
: temporals temporal
  { }
  |
  { }
;

temporal
: TEMPORAL formal_name defaults where response
  { }
;
! Temporal events are triggered at absolute times,
! in terms of the local clock of the actor.
! The "where" describes the triggering conditions
! in terms of TIME, PERIOD, and DELAY.

transactions
: transactions transaction
  { }
  |
  { }
;

transaction
: TRANSACTION actual_name '=' action_list where foreach
  { }
;
! Transactions are atomic.
! The where clause can specify timing constraints.

```

```

action_list
: action_list ';' action    %prec SEMI    ! sequence
  { }
! action
  { }
;

action
: action action    %prec STAR    ! unordered set of actions
  { }
! IF alternatives FI    ! choice
  { }
! DO alternatives OD    ! repeated choice
  { }
! actual_name    ! a normal message or subtransaction
  { }
! EXCEPTION actual_name    ! an exception message
  { }
;

alternatives
: alternatives OR guard action_list
  { }
! guard action_list
  { }
;

guard
: WHEN expression_list ARROW
  { }
!
  { }
;

concepts
: concepts concept
  { }
!
  { }
;

concept
: CONCEPT formal_name ':' expression defaults where
  ! constants
  { }
! CONCEPT formal_name '(' formals ')' defaults
                                     VALUE '(' formals ')' where

```

```

        ! functions, defined with preconditions and postconditions
    ( )
;

optional_formal_name

: formal_name
{ }
|
{ }
;

formal_name
: NAME '{' formals '}'
{ formal_name.name = NAME.%text;
  formal_name.generic_parameters = ["generic\n\t",
    formals.generic_parameters, ";\n"]; }
| NAME
{ formal_name.name = NAME.%text;
  formal_name.generic_parameters = ""; }
;

formal_arguments
: '(' formals ')'
{ formal_arguments.input_parameters = formals.input_parameters;
  formal_arguments.output_parameters = formals.output_parameters;
  formal_arguments.num_output_variables =
    formals.num_output_variables;
  formal_arguments.return_type = formals.return_type; }
|
{ formal_arguments.input_parameters = "";
  formal_arguments.output_parameters = "";
  formal_arguments.return_type = ""; }
;

formals
: field_list restriction
{ formals.input_parameters = field_list.input_parameters;
  formals.output_parameters = field_list.output_parameters;
  formals.generic_parameters = field_list.generic_parameters;
  formals.num_output_variables = field_list.num_output_variables;
  formals.return_type = field_list.return_type; }
;

```

```

field_list
: field_list ',' type_binding
  { field_list[1].input_parameters =
    [field_list[2].input_parameters, "; ",
    type_binding.input_parameters];
    field_list[1].generic_parameters =
    [field_list[2].generic_parameters, "; ",
    type_binding.generic_parameters];
    field_list[1].output_parameters =
    [field_list[2].output_parameters, "; ",
    type_binding.output_parameters];
    field_list[1].num_output_variables =
    field_list[2].num_output_variables + 1;
    field_list[1].return_type = type_binding.return_type; }
| type_binding
  { field_list.input_parameters = type_binding.input_parameters;
    field_list.output_parameters = type_binding.output_parameters;
    field_list.generic_parameters = type_binding.generic_parameters;
    field_list.num_output_variables = 1;
    field_list.return_type = type_binding.return_type; }
;

type_binding
: name_list ':' expression
  { type_binding.input_parameters =
    [name_list.parameter_name, ": in ",
    expression.parameter_type];
    type_binding.generic_parameters =
    [name_list.parameter_name, ": ",
    expression.parameter_type];
    type_binding.output_parameters =
    [name_list.parameter_name, ": out ",
    expression.parameter_type];
    type_binding.return_type = expression.return_type; }
| '$' NAME ':' expression
  { }
;

name_list
: name_list NAME
  { name_list[1].parameter_name = [name_list[2].parameter_name, ", ",
    NAME.$text]; }
| NAME
  { name_list.parameter_name = NAME.$text; }
;

```

```

restriction
    : SUCH expression_list
      { }
    |
      { }
    ;

optional_actual_name
    : actual_name
      { optional_actual_name.exception_declarations =
        ["\t",actual_name.exception_declarations, ": exception;\n"]; }
    |
      { optional_actual_name.exception_declarations = ""; }
    ;

actual_name
    : NAME '(' actuals ')'
      { }
    | NAME %prec SEMI ! must have a lower precedence than '('
      { actual_name.parameter_type = NAME.%text;
        actual_name.return_type = NAME.%text;
        actual_name.exception_declarations = NAME.%text; }
    ;

actuals
    : actuals ',' arg %prec COMMA
      { }
    | arg
      { }
    ;

arg
    : expression
      { }
    | pair
      { }
    ;

expression_list
    : expression_list ',' expression %prec COMMA
      { }
    | expression %prec COMMA
      { }
    ;

```



expression

```
: QUANTIFIER '(' formals BIND expression ')'
{ }
| actual_name      ! variables and constants
  { expression.parameter_type = actual_name.parameter_type;
    expression.return_type = actual_name.return_type;
  }
| expression '(' actuals ')'      ! function call
{ }
| expression '@' actual_name      ! expression with explicit type cast
{ }
| NOT expression    %prec NOT
{ }
| expression AND expression    %prec AND
{ }
| expression OR expression    %prec OR
{ }
| expression IMPLIES expression %prec IMPLIES
{ }
| expression IFF expression    %prec IFF
{ }
| expression '=' expression      %prec LE
{ }
| expression '<' expression      %prec LE
{ }
| expression '>' expression      %prec LE
{ }
| expression LE expression      %prec LE
{ }
| expression GE expression      %prec LE
{ }
| expression NE expression      %prec LE
{ }
| expression NLT expression      %prec LE
{ }
| expression NGT expression      %prec LE
{ }
| expression NLE expression      %prec LE
{ }
| expression NGE expression      %prec LE
{ }
| expression EQV expression      %prec LE
{ }
| expression NEQV expression      %prec LE
{ }
| '-' expression    %prec UMINUS
{ }
```

```

| expression '+' expression          %prec PLUS
| { }
| expression '-' expression          %prec MINUS
| { }
| expression '*' expression          %prec MUL
| { }
| expression '/' expression          %prec DIV
| { }
| expression MOD expression          %prec MOD
| { }
| expression EXP expression          %prec EXP
| { }
| expression U expression %prec U
| { }
| expression APPEND expression      %prec APPEND
| { }
| expression IN expression          %prec IN
| { }
| '*' expression %prec STAR
| ! *x is the value of x in the previous state
| { }
| '$' expression %prec DOT
| ! $x represents a collection of items rather than just one
| ! s1 = {x, $s2} means s1 = union({x}, s2)
| ! s1 = [x, $s2] means s1 = append([x], s2)
| { }
| expression RANGE expression      %prec RANGE
| ! x in [a .. b] iff x in {a .. b} iff a <= x <= b, [a .. b] is
| ! sorted in increasing order
| { }
| expression '.' NAME %prec DOT
| { }
| expression '[' expression_list ']' %prec DOT
| { }
| '(' expression ')'
| { }
| '(' expression NAME ')' ! expression with units of measurement
| ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS MINUTES
| !
| ! HOURS DAYS WEEKS
| { }
| TIME ! The current local time, used in temporal events
| { }
| DELAY ! The time between the triggering event and
| ! the response
| { }
| PERIOD ! The time between successive events of this type
| { }

```

```

| literal      ! literal with optional type id
  ( )
| '?'          ! An undefined value to be specified later
  ( )
| '!'          ! An undefined and illegal value
  ( )
| IF expression THEN expression middle_cases ELSE expression FI
  ( )
;

middle_cases
: middle_cases ELSE_IF expression THEN expression
  ( )
|
  ( )
;

literal
: INTEGER_LITERAL
  ( )
| REAL_LITERAL
  ( )
| CHAR_LITERAL
  ( )
| STRING_LITERAL
  ( )
| '#' NAME      ! enumeration type literal
  ( )
| '[' expressions ']' ! sequence literal
  ( )
| '(' expressions ')' ! set literal
  ( )
| '(' formals BIND expression ')' ! set literal
  ( )
| '(' expressions ';' expression ')' ! map literal
  ( )
| '[' pair_list ']' ! tuple literal
  ( )
| '(' NAME BIND expression ')' ! union literal
  ( )
;

! relation literals are sets of tuples

expressions
: expression_list
  ( )
|

```

```

        ( )
    ;

pair_list
: pair_list ',' pair
  ( )
  | pair
  ( )
  ;

pair
: name_list BIND expression
  ( )
  ;

```

## **APPENDIX C**

### **ADACI USER'S GUIDE**

#### **A. INTRODUCTION**

The purpose of this user's guide is to enable a user to generate an Ada specification from a valid Spec specification via the concrete interface generation system. This guide assumes an executable version of the concrete interface generation system has been installed on your system. The user's guide also assumes the reader is familiar with Spec and Ada.

#### **B. SPEC INPUT**

The user must provide a valid Spec input file for the concrete interface generation system.

The concrete interface generation system has not been fully implemented. For the current version of the system, "valid Spec input" is a file containing one Spec FUNCTION module. The current version ignores the following non-terminals and their productions:

- types
- machines
- definitions
- pragmas
- inherits
- hide

- renames
- imports
- export

### **C. GETTING STARTED**

Make sure that the command "adaci" is in your name space by entering the command "**which adaci**" at the UNIX prompt. If it responds with a pathname, you are ready to use the system.

If you get an error message that starts with "no adaci in ...," you should edit your .cshrc file to add the path for the Spec tools to the path variable in your unix shell. On suns2, this path should be "/usr/spectools" and the line added to your .cshrc file should look like the following:

```
set path = ($path /usr/spectools)
```

### **D. SYSTEM OPERATION**

Operation of the concrete interface generation system is very simple. Place the file containing a valid Spec specification in a file named "spec\_file\_name" in your current working directory. From the UNIX prompt, enter the command:

```
adaci spec_file_name > ada_file_name
```

The concrete interface generation system will store the generated Ada specification in the file "ada\_file\_name." The Ada specification will be displayed on the terminal screen if the "> ada\_file\_name" is

omitted. The command `adaci` can also be used as a filter in a UNIX pipe.

### E. SAMPLE EXECUTION

To generate the Ada specification for the Spec FUNCTION "square\_root" using the concrete interface generation system, follow these steps.

1. Store the Spec FUNCTION "square\_root" shown in Figure C.1 in a file named "sqrt.s".

```
FUNCTION square_root(precision:float SUCH THAT precision > 0.0
  MESSAGE(x: float)
    WHEN x >= 0.0
      REPLY(y: float)
        WHERE y > 0.0, approximates(y * y, x)
      OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: float)
    VALUE(b: boolean)
      WHERE b <=> abs(r1 - r2) <= abs(r2 * precision)
END
```

Figure C.1. Generic Example, Input: Spec

2. At the UNIX prompt enter the command:

**`adaci sqrt.s > sqrt.s.a`**

3. At the UNIX prompt enter the command: **`more sqrt.s.a`**
4. The Ada specification of Figure C-2 will be displayed on the terminal screen.

```
generic
    precision: float;
package square_root_pkg is
    function name(x: in float) return float;
    imaginary_square_root: exception;
end square_root_pkg;
```

**Figure C.2. Generic Example, Output: Ada Specification**



## LIST OF REFERENCES

1. Berzins, V., and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1990.
2. Berzins, V. and Luqi, "An Introduction to the Specification Language SPEC," *IEEE Software*, v. 7, no. 2, March 1990.
3. Kopas, R. G., *A Students Guide to Spec*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.
4. Knuth, D. E., "Semantics of Context-Free Languages," *Mathematical Systems Theory*, v. 2, pp. 127-145, November 1967.
5. Herndon, R., and Berzins, V., "The Realizable Benefits of a Language Prototyping Language," *IEEEETSE*, v. SE-14, no. 6, pp. 803-809, June 1988.
6. University of Minnesota Computer Science Technical Report 85-37, *The Incomplete AG User's Guide and Reference Manual*, by R. M. Herndon, Jr., October 1985.
7. Bell Laboratories Computer Science Technical Report 39, *Lex A Lexical Analyzer Generator*, by M. Lesk and E. Schmidt, October 1975.
8. Johnson, S., *YACC—Yet Another Compiler Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.
9. Herndon, R., *Attribute Grammar Systems for Prototyping Translators and Languages*, Ph. D. Dissertation, University of Minnesota, 1988.
10. Kopas, R. G., *The Design and Implementation of a Specification Language Type Checker*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.
11. Altizer, C., *Implementation of a Language Translator for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.
12. DePasquale, G., *The Design and Implementation of a Automated Unit Verification System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1990.

13. Weigand, J., *Design and Implementation of a Pretty Printer for the Functional Specification Language Spec*, M.S. Thesis, Naval Post-graduate School, June 1988.
14. Department of Defense Directive 3405.1, April 2, 1987.

## INITIAL DISTRIBUTION LIST

	<u>No. Copies</u>
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943-5000	1
4. Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
5. Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
6. National Science Foundation Division of Computer and Computation Research Washington, DC 20550	1
7. Office of the Chief of Naval Operations Code OP-941 Washington, DC 20350	1
8. Office of the Chief of Naval Operations Code OP-945 Washington, DC 20350	1
9. Commander, Naval Data Automation Command Washington Navy Yard Washington, DC 20374-1662	1

10. Commanding Officer 1  
Code 5150  
Naval Research Laboratory  
Washington, DC 20375-5000
11. Defense Advanced Research Projects Agency (DARPA) 1  
Integrated Strategic Technology Office (ISTO)  
1400 Wilson Boulevard  
Arlington, VA 22209-2308
12. Defense Advanced Research Projects Agency (DARPA) 1  
Director, Naval Technology Office  
1400 Wilson Boulevard  
Arlington, VA 22209-2308
13. Defense Advanced Research Projects Agency (DARPA) 1  
Director, Prototype Projects Office  
1400 Wilson Boulevard  
Arlington, VA 22209-2308
14. Dr. R. M. Carroll (OP-01B2) 1  
Chief of Naval Operations  
Washington, DC 20350
15. Dr. Ted Lewis 1  
Editor-in-Chief, IEEE Software  
Oregon State University  
Computer Science Department  
Corvallis, OR 97331
16. Dr. R. T. Yeh 1  
International Software Systems Inc.  
12710 Research Boulevard, Suite 301  
Austin, TX 78759
17. Professor D. Berry 1  
Department of Computer Science  
University of California  
Los Angeles, CA 90024
18. Ada Joint Program Office 1  
OUSDRE(R&AT)  
The Pentagon  
Washington, DC 20300

19. Dr. Van Tilborg 1  
Office of Naval Research  
Computer Science Division, Code 1133  
800 N. Quincy Street  
Arlington, VA 22217-5000
20. Dr. R. Wachter 1  
Office of Naval Research  
Computer Science Division, Code 1133  
800 N. Quincy Street  
Arlington, VA 22217-5000
21. Dr. L. Belady 1  
Software Group, MCC  
9430 Research Boulevard  
Austin, TX 78759
22. Dr. C. V. Ramamoorthy 1  
University of California  
Department of Electrical Engineering and Computer Science  
Computer Science Division  
Berkeley, CA 94720
23. Dr. Earl Chavis (OP-162) 1  
Chief of Naval Operations  
Washington, DC 20350
24. Dr. Alfs Berztiss 1  
University of Pittsburgh  
Department of Computer Science  
Pittsburgh, PA 15260
25. Mr. Joei Trimble 1  
1211 South Fern Street, C107  
Arlington, VA 22202
26. Dr. Luqi 1  
Code CS/Lq  
Computer Science Department  
Naval Postgraduate School  
Monterey, CA 93943-5000
27. Dr. V. Berzins 2  
Code CS/Bz  
Computer Science Department  
Naval Postgraduate School  
Monterey, CA 93943-5000